Idle In Transaction Problem

Many problems that arise from the database can be attributed to database transactions that have not been completed propertly. This can cause the connection pool to fill to the limit and to cause other database operations such as VACUUM to fail to complete. Connection pooled transactions should read "idle" next to their process name, while transactions which have not yet completed successfully usually read "idle in transaction". Other messages you are likely to see include "SELECT waiting", which is the most serious problem, because this means that an HTTP request was made but the SELECT to get the data never completed - probably the user got fed up of waiting and went away.

The transactions viewable on the process queue which are annotated "idle in transaction" are transactions which have taken out a lock on a table or tables, and have then failed to complete for whatever reason (we'll come to that in a moment), so they have hung, taking up a transaction thread and maintaining their lock on the database. Therefore, if such transactions do not ever die, problems can ensue.

So the objective is to get rid of these "idle in transaction" connections. There are 2 solutions, the first of which is just a workaround:

1) Restarting the database prior to an operation like vacuum will kill all existing transactions. This is decisive and swift, but runs the risk of interrupting real transactions as well as the duds.

2) Find and fix the places in the code where transactions are not safely completed. This is surprisingly hard, because although the application framework initialises and closes transactions at start and finish, there are a variety of things that can happen causing the transaction to fail. Principally, exceptions that occur at the database level can have unfortunate consequences for the state of the transaction, so this needs to be looked at. In addition, any situation which causes a Context object to go out of scope (which can happen if additional context objects are created in the course of the processing) can leave a hanging transaction.

Contents

Determining locked tables

There is no obvious way of tracking down all these situations, but we can offer a query which will allow us to generate a "signature" for the requests that cause the problems. This is done using the following SQL:

```
SELECT relation, transaction, pid, mode, granted, relname
FROM pg_locks
INNER JOIN pg_stat_user_tables
ON pg_locks.relation = pg_stat_user_tables.relid
WHERE pg_locks.pid='pid';
```

This will give us a list of table names that a given process (identified by its pid) has locked and not released. This will hopefully allow us to identify which requests are causing the problem. Therefore, if you run this query, you should be able to see if this is your own code (accessing, for example, your tables, or a set of tables that one of your operations uses) or whether it is a DSpace bug. You should post this signature to the mailing list if you can't debug it yourself.

For example, this is an output from the above query:

relation	transaction	pid	mode	granted	relname
18034		4909	AccessShareLock	t	group2groupcache
18113		4909	AccessShareLock	t	metadatavalue
18046		4909	AccessShareLock	t	item
18008		4909	AccessShareLock	t	eperson
18248		4909	AccessShareLock	t	handle
18235		4909	AccessShareLock	t	epersongroup2eperson

So process 4909 has AccessShareLocks on 6 tables, and these 6 tables should help us identify where to look for the bug. In this case it is particularly unclear, but a process which uses groups, epersons, items handles and metadata is probably a deposit process.

Logging queued queries

To see exactly which queries are causing the errors, we can go a little further. In postgres.conf, set the parameter:

```
stats_command_string = true
```

When reinitialised, this will store the SQL waiting to be executed on each postgres process in the table pg_stat_activity:

```
SELECT * FROM pg_stat_activity;
```

which produces something like:

datid	datname	procpid	usesysid	usename	current_query	query_start
17142	dspace	1107	1	dsprd	<IDLE>	2007-06-11 11:21:36.147423 +01
17142	dspace	1401	1	dsprd	<IDLE>	2007-06-11 11:22:41.730301 +01
17142	dspace	25163	1	dsprd	<IDLE>	2007-06-11 11:21:36.097556 +01
17142	dspace	27973	1	dsprd	update history set creation_date = '...'	2007-06-11 11:25:39.687193 +01
17142	dspace	916	1	dsprd	<IDLE>	2007-06-11 11:21:32.759784 +01

Combining queued queries with locks

A useful query acquired from a user comment on the Postgres documentation page:

http://www.postgresql.org/docs/current/interactive/monitoring-locks.html

```
select pg_class.relname, pg_locks.transaction, pg_locks.mode, pg_locks.granted as "g",
substr(pg_stat_activity.current_query,1,30), pg_stat_activity.query_start,
age(now(),pg_stat_activity.query_start) as "age", pg_stat_activity.procpid
from pg_stat_activity,pg_locks
left outer join pg_class on
(pg_locks.relation = pg_class.oid)
where pg_locks.pid=pg_stat_activity.procpid
order by query_start;
```

An other query from Jeff Davis shows any queries that are waiting on a lock, and the
query that currently holds the lock on which those queries are waiting: (not included here for brevity)

http://groups.google.com/group/pgsql.general/browse_thread/thread/10f61707d242a308?pli=1

Workaround: Killing "Idle in Transaction" processes with crontab

Cory Snavely offered this workaround on dspace-tech:

For the moment, I have installed a dirty little crontab entry that runs
this on the minute:

```
/usr/bin/test `/usr/bin/pgrep -f 'idle in transaction' | \
/usr/bin/wc \-l ` \-gt 20 && /usr/bin/pkill \-o \-f 'idle in transaction'
```

In English: every minute, if there are more than 20 "idle in
transaction" Postgres processes, it kills the oldest one.

Wally Grotophorst's OS X version:

```
/bin/test `/usr/local/bin/pgrep -f '127.0.0.1' | \
/usr/bin/wc \-l ` \-gt 20 && /usr/local/bin/pkill \-n \-f '127.0.0.1'
```